

1. 2/2C/2009 (1)

Ejercicio 1

1. Escribir (en castellano) la propiedad que hace que un ABB sea AVL (o sea, el invariante).
2. Proponer un algoritmo que verifique en tiempo lineal si un ABB cumple el invariante de AVL.
3. Presentar un ejemplo de árbol AVL en el cual el borrado de un elemento dé lugar a más de una rotación.
1. Para todo nodo x perteneciente a T , siendo T un AVL, se debe cumplir que $!Hoja?(x) \implies |altura(izq(x)) - altura(der(x))| \leq 1$, es decir que la diferencia entre el sub-árbol derecho e izquierdo de cualquier nodo x del árbol debe ser menor o igual a 1.

2. Sea $dameAlturaAVL(AVL : T)$ una función que devuelve la altura del AVL si cumple con el invariante y devuelve ∞ en caso contrario.

El caso base de esta función se dará cuando $Nil?(T) = True$, en donde trivialmente cumplirá con el invariante y su altura será cero. Luego, para el caso recursivo verificaremos la propiedad $|dameAlturaAVL(izq(x)) - dameAlturaAVL(der(x))| < 2$, en donde $\infty - \infty < 2 \equiv False$. Si la propiedad es falsa devolveremos ∞ mientras que si es verdadera devolveremos $\max\{dameAlturaAVL(izq(x)), dameAlturaAVL(der(x))\} + 1$.

La recurrencia de esta función será de la forma $T(n) = 2 \cdot T(2/n) + \Theta(1)$, por lo que $a = 2$, $b = 2$ y $f(n) \subseteq \Theta(1)$. Luego, tendremos que $n^{\log_b(a)} = n^{\log_2(2)} = n^1$, que con un $\epsilon = 1 > 0$ puede dejarse como $n^{1-\epsilon} = n^0 = 1$ y como $f(n) \subseteq \Theta(1)$ en particular $f(n) \subseteq O(1)$, por lo que caeremos en el primer caso del teorema maestro, lo que implicará que $T(n) \subseteq \Theta(n)$.

3. 3 nodos, P , Q y R , están establecidos de la forma $izq(Q) = h$, $der(Q) = P$, $izq(P) = R$, $der(P) = h - 1$, $izq(R) = der(R) = h - 1$, por lo que sus FDB son $Fdb(Q) = 1$, $Fdb(P) = -1$ y $Fdb(R) = 0$. Finalmente si eliminamos algún elemento que pertenezca al sub-árbol $izq(Q)$, el FDB de Q pasará a valer 2 y el caso será el mismo que se da en LR luego de la inserción.

Ejercicio 2

1. Cuándo decimos que una función no es congruente con la igualdad observacional?
 - a) Cuando diferencia más instancias que los observadores básicos.
 - b) Cuando es redundante con respecto a los observadores básicos.
 - c) Ambas.
 - d) Ninguna.
2. Cual es la diferencia entre tipo y género (por ejemplo, entre NAT y nat)?
3. Qué criterio utilizaría para definir si una operación puede ser generador o ser otra operación?
 1. Cuando diferencia más instancias que los observadores básicos.
 2. El tipo es el conjunto de operaciones y axiomas que componen a un TAD específico mientras que el género es el nombre con el que representamos las instancias de un TAD.
 3. Esto depende bastante del contexto en donde estemos. En los casos en donde podremos estar entre hacer una función como generador o como otra operación son aquellos en donde la operación nos debe devolver una nueva instancia del TAD modificada. Si decidimos hacer un generador estaremos agregando una "capa" mas para informar de una nueva situación en donde nos encontramos, de forma contraria si decidimos hacer una otra operación posiblemente estemos quitando una capa de las agregadas anteriormente.

Para dar un ejemplo, supongamos el ejercicio de fila del banco que se presenta en la guía. En este ejercicio inicialmente tendremos una fila simple a la que solamente pueden llegar personas y ser atendida, y a medida que nos desplazamos por los puntos del ejercicio se necesita incrementar el nivel de expresión del modelo por lo que se agregan nuevas operaciones, siendo una de estas "retirarse". La operación retirarse puede ser implementada como un generador recursivo o como una otra operación. En el caso de ser un generador estaremos representando cuando una persona se retira de la fila agregando mas información a la instancia, mientras que si la implementamos

como una otra operación podremos buscar la llegada de la persona a la fila para eliminarla de la instancia. Si bien observacionalmente los resultados son idénticos (si no nos interesa saber quien se fue sin ser atendido), en el segundo caso tendremos menos información ya que la instancia construida por los generadores será igual en los casos cuando alguien se retira de la fila y en el caso de que nunca estuvo en la misma.

Por esto mismo, al decidir si una operación debe ser generador o no, bajo mi criterio, deberemos pensar en posibles modificaciones a futuro del TAD, ya que si alguna vez deseamos conocer cierta información con respecto a la aplicación de la operación, si la definimos como un generador podremos obtenerla fácilmente agregando algun observador, mientras que si la definimos de la otra forma deberemos modificar mas intensivamente el TAD. La desventaja de agregar un generador mas al TAD es que deberemos crear un axioma mas por cada observador básico que tengamos.

Ejercicio 3

Considere el TAD MinColaDePrioridad enriquecido con la operación DisminuirPrioridad(p, x) que, dado un elemento de clave p , le disminuye la prioridad a p en x unidades.

1. Discutir las modificaciones a realizar a las representaciones clásicas de colas de prioridad para incorporar eficientemente la operación DisminuirPrioridad(p, x) y describir brevemente el algoritmo para esa operación, y su complejidad.
2. Y si además se quiere incorporar la operación AumentarPrioridad(p, x)?

Respuesta:

1. Las dos implementaciones clásicas para una cola de prioridad son la lista enlazada y max-heap. Si utilizamos una lista enlazada encontrar el elemento a modificar su prioridad a partir de su clave nos llevara $O(n)$ ya que en el peor caso deberemos recorrer toda la lista, luego ubicarlo en su nueva posición nos llevara nuevamente $O(n)$ en el peor caso. El algoritmo en este caso no tiene mucha dificultad.

En el caso del heap implementado sobre un arreglo, podremos recorrer el arreglo para encontrar el elemento y luego de reducir su prioridad ejecutar "bajar", lo que volverá la operación $O(n + \lg(n)) \subseteq O(n)$. Sin embargo podremos hacerlo mejor si agregamos a la estructura de representación algo que nos permita encontrar mas rápidamente el elemento con la clave. Para esto podremos agregar un *Dicc(clave, puntero(nodo))*, en donde los punteros apunten a los nodos del heap, que puede estar representado de diferentes formas dependiendo el tipo de datos de la clave:

- a) Para cualquier tipo de dato al que pertenezca la clave siempre podremos utilizar una tabla de hash como representación para el diccionario, la cual nos dará una complejidad de $O(\lg(n))$ en el caso promedio y $O(n)$ en el peor caso.
- b) Si el dato de la clave es comparable, es decir tiene un orden, podremos utilizar un AVL como representación del diccionario, lo que nos dejara el algoritmo en una complejidad de $O(\lg(n))$ en el peor caso ya que la búsqueda.
- c) Si el dato de la clave es un string, podremos usar el caso anterior (con un orden lexicográfico), o podremos utilizar un trie. Esto nos dejara con una complejidad resultante de $O(|p| + \lg(n))$ en donde $|p|$ es la longitud de la clave. Si podemos acotar la longitud de las claves por alguna constante, entonces nos quedara que la complejidad en el peor caso es de $O(\lg(n))$.

Las mismas 3 variantes se podrán implementar para encontrar un elemento mas velozmente en el caso de la lista enlazada, aunque no mejorara la complejidad del algoritmo ya que ubicarlo en su nueva posición todavía tendrá costo $O(n)$.

2. Para la operación de AumentarPrioridad, utilizaremos la misma estructura secundaria para ubicar el elemento mas rápidamente y luego de haber modificado su valor haremos un procedimiento similar al que realizamos durante la inserción de un valor nuevo en el heap. Sea x el nodo que habremos modificado su valor de forma incremental, podremos estar rompiendo el invariante del sub-heap que tiene su raíz en el padre de x . Para reestablecer el mismo compararemos el valor de x con el valor de su padre. Si el valor de x es mayor al valor de su padre $x.p$, haremos un intercambio de posiciones en el heap por lo que ahora $x \leftarrow x.p$. Repetiremos esta operación hasta que el invariante del heap este satisfecho. La operación a lo sumo llevara $O(\lg(n))$ ya que a lo sumo llegaremos a la raíz y la altura máxima del árbol es de $\lg(n)$, lo que nos deja con los mismos costos que DisminuirPrioridad a la operación AumentarPrioridad.

Ejercicio 4

Responda justificando.

1. Tiene sentido programar una implementación del invariante de representación?
 2. Y la función de abstracción?
 3. Puede el invariante ser restricción de una función auxiliar? Debe serlo?
 4. De dos ejemplos de relaciones entre invariante de representación y complejidad de los algoritmos.
1. Bajo mi punto de vista, solamente tiene sentido implementarlo si en la implementación tenemos algún constructor que usa directamente una instancia de la estructura de representación para construir la instancia del modulo. De esta forma con un invariante de representación implementado podríamos verificar que la instancia de la estructura que nos pasan como parámetro es una instancia valida. Otro caso en donde puede ser útil la implementación del invariante es para verificar que las operaciones funcionen correctamente exponiéndolas a distintos escenarios, es decir para testear nuestro modulo. Igualmente habrá que tener en cuenta que en el momento de verificar la validez del predicado del invariante podrá tomar mucho tiempo por la naturaleza del mismo, ya que puede contener cuantificadores y podremos caer en un problema intratable.
 2. No tiene sentido ya que la función de abstracción nos devolverá una instancia del TAD que describe al modulo y que no nos servirá para nada en la implementación, ya que la misma servirá para demostrar formalmente que nuestras operaciones exportadas realizan lo que el TAD que explica nuestro modulo especifica.
 3. El invariante de representación esta implícito en todas las operaciones exportadas del modulo, no aplicara sobre las funciones que no se exportan del mismo, incluyendo las funciones auxiliares.
 4. Un ejemplo es entre el invariante de un ABB y un AVL. El invariante del ABB nos dice que todo nodo en el sub-arbol izquierdo de un nodo x debe ser menor o igual a x y que todo nodo en el sub-arbol derecho de x debe ser mayor a x , lo que nos da una complejidad de $O(\lg(n))$ en el caso promedio y $O(n)$ en el peor caso. En cambio, si agregamos el predicado del invariante del AVL que dice que entre todo sub-arbol derecho e izquierdo de un nodo x no puede haber una diferencia total de mas de 1 unidad de alto, automáticamente las complejidades pasan a ser todas $O(\lg(n))$ en el peor caso, si es que implementamos los algoritmos de forma coherente. Es decir que de cierta forma el invariante de representación no solo obliga a las operaciones a mantener la coherencia de la estructura sino que las obliga a tener una complejidad.

Ejercicio 5

Dada la siguiente frase construida sobre el alfabeto $\{ESP, N, O, S, T, R, M, L, C, Z\}$: NOSOTROS NO SOMOS COMO LOS OROZCOS (El enunciado fue cambiado por el final del 2/2C/2009, es exactamente el mismo ejercicio)

1. Construir un código de Huffman para los caracteres de la frase (dar el árbol o la tabla de códigos).
 2. Cuántos bits se ganan en la codificación de la frase con respecto a la utilización de un código de longitud fija?
 3. Discutir aspectos relacionados con la implementación del algoritmo de Huffman (qué estructuras de datos usaría para hacerlo en forma eficiente? qué complejidad resultaría su algoritmo?).
1. Construcción siguiendo el algoritmo de Huffman.
 2. Teniendo 10 caracteres distintos que codificar, necesitaremos al menos 4 bits para hacerlo ya que $2^3 < 10 < 2^4$, por lo que tendremos $34 \cdot 4 = 136$ bits para codificar el texto contra 98 utilizando un código variable con prefijos óptimos, lo que nos deja una ganancia de 38 bits.
 3. Para construir el árbol de Huffman propiamente dicho utilizaría nodos cuyos atributos sean un valor, punteros a sus hijos izquierdo y derecho y un caracter (en el caso de que corresponda alguna codificación). Luego, en el algoritmo utilizaría un min-heap de nodos, ordenados a partir de su valor. El algoritmo comenzaría construyendo un arreglo A de nodos de longitud $n = |C|$, en donde cada uno correspondería a un caracter que necesita ser codificado y su valor seria la frecuencia de dicho caracter. Luego, se aplicara Heapify al arreglo A para convertirlo en un min-heap Q en $O(n)$.

Durante el ciclo del algoritmo de Huffman, en cada paso se obtendrán los dos nodos x, y con menor valor, se construirá un nuevo nodo z , se asignaran como hijos de z a x, y , de la forma $z.der = x$, $z.izq = y$ y se asignara el valor de z de la forma $z.valor = x.valor + y.valor$. Luego se insertara z en Q . Podremos acotar la cantidad de elementos en Q por n , ya que en la primera iteración es en donde tendrá mas elementos, y por lo tanto el tiempo insumido en un paso sera de $O(3lg(n)) \subseteq O(lg(n))$. Nuestro árbol tendrá $n = |C|$ hojas, por lo que tendrá $n - 1$ nodos internos, que sera la cantidad de pasos anteriormente descriptos que se ejecutaran, dejando al bucle con una complejidad total de $O(n \cdot lg(n))$ y al algoritmo como $O(n \cdot lg(n) + n) \subseteq O(n \cdot lg(n))$.

2. 2/2C/2009 (2)

Ejercicio 1

Daban una especificación de un TAD, y debían marcarse y corregirse los errores, especificando porque eran errores.

Ejercicio 2

Dado dos array de números enteros A y B (pueden haber elementos repetidos), de tamaño n y m respectivamente, proponer algoritmos para calcular la intersección de manera EFICIENTE si

1. $n = m$
 2. $n \sim m$, m es chico
 3. $n \sim m$, m es grande
1. Ordenaremos los dos arreglos por el método $O(n \cdot \lg(n))$ de nuestra preferencia, luego haremos una comparación de la misma forma que merge lo hace, con la diferencia que guardaremos en una lista nueva los elementos que sean iguales. Finalmente devolveremos la lista. La complejidad de ordenar ambos arreglos sera de $O(n \cdot \lg(n) + m \cdot \lg(m))$ mientras que la comparación e inserción en el caso de tener elementos iguales tomara tiempo $O(n+m)$. Finalmente nos dará una complejidad total de $O(n \cdot \lg(n) + m \cdot \lg(m) + (n + m)) \subseteq O(4n \cdot 2\lg(n)) \subseteq O(n \cdot \lg(n))$. Otra forma de realizar esto es, cargar todos los m elementos en un *Conj* representado con un *AVL*, lo que llevara tiempo $O(m \cdot \lg(m))$. Luego, se comprobara la existencia en el conjunto para cada uno de los n elementos y de existir, se lo agregara a la lista que sera devuelta. El tiempo de este algoritmo sera de $O(m \cdot \lg(m) + n \cdot \lg(m)) \subseteq O((m + n) \cdot \lg(m)) \subseteq O(2n \cdot \lg(n)) \subseteq O(n \cdot \lg(n))$.
 2. Si utilizamos el ultimo algoritmo propuesto tendremos una complejidad de $O((m + n) \cdot \lg(m))$, si tomamos en consideración que m es chico y n será mucho mas grande, esto puede ser fácilmente acotado por $O(n)$.
 3. **Mismo algoritmo?**

Ejercicio 3

Proponer una estructura de datos que permita implementar un diccionario de palabras de manera de que la búsqueda de un termino específico sea eficiente. Además, se debe poder buscar todas las palabras que tengan un cierto largo y un cierto prefijo en común. Dar las complejidades de estas dos operaciones, y de la de carga del diccionario (dado un texto, cargar todas sus palabras)

La estructura de datos sera un $Dicc_{AVL}(Int : clave, Dicc_{Trie} : valor)$. Durante la inserción de una palabra l se utilizara su longitud $|l|$ como clave del diccionario $Dicc_{AVL}$, y luego se la insertará en el $Dicc_{Trie}$ que se encuentre en el significado de la clave. Si la clave $|l|$ resulta nueva, se creara un nuevo *Trie*. Esto nos dejara con una complejidad de $O(\lg(|D|) + |l|)$ para la operación de inserción en donde D será el diccionario total de palabras. Luego, la operación especial tomara tiempo $O(\lg(|D|) + |l| \cdot |P|)$, en donde P es el conjunto de palabras que comienza con el prefijo que se dio, esto es así ya que dentro del trie podremos ir hasta el nodo que contenga el final del prefijo, y desde allí explorar todas las ramas posibles, lo que nos tomara tiempo $O(|l| \cdot |P|)$, en donde $|l|$ en este caso hará referencia a la longitud que nos den por parámetro. Sea un texto W , y sea w_i la palabra en la posición i del texto, la complejidad de la carga del diccionario sera de $O(\sum_{i=1}^{|W|} \lg(|D|) + |w_i|)$, en donde en el peor caso cargaremos una palabra diferente por cada vez, lo que incrementara nuestro diccionario en una unidad por cada palabra dejando así $O(\sum_{i=1}^{|W|} \lg(i) + |w_i|)$ lo que puede ser acotado por $O(|W| \cdot \lg(|W|) + |W| \max_{i \in |W|} |w_i|)$.

En el caso de poder acotar las palabras por una longitud máxima, podremos reemplazar el AVL por un arreglo de c posiciones, en donde c sera la longitud máxima. Luego la inserción pasará a ser $O(1)$, y la operación especial pasara a ser $O(|P|)$. En este caso la carga del diccionario tendrá una complejidad de $O(|W| \cdot \lg(|W|) + c \cdot |W|) \subseteq O(|W| \cdot \lg(|W|))$ para un texto con suficientemente grande cantidad de palabras.

Ejercicio 4

Mismo ejercicio que en 2/2C/2009 (1)

Ejercicio 5

Responda justificando. Tiene sentido programar una implementación del invariante de representación? Y la función de abstracción? De dos ejemplos de relaciones entre invariante de representación y complejidad de los algoritmos.

Mismo ejercicio que el ejercicio 4 de 2/2C/2009 (1)

3. 1C/2010

Ejercicio 1

Sea S una secuencia de n claves enteras. No hacemos ninguna hipótesis sobre el rango de valores que cubren las claves, que puede ser arbitrariamente grande. Sin embargo, sabemos que las claves pueden tomar $\lfloor \log(n) \rfloor$ valores distintos. Por ejemplo, para $n = 8$ una secuencia con esas características podría ser $\langle 349, 12, 12, 102, 349, 12, 102, 102 \rangle$.

1. Desarrollar un algoritmo para ordenar S en tiempo $o(n \cdot \lg(n))$ (o sea ESTRICTAMENTE menor que $O(n \cdot \lg(n))$), explicarlo y analizar su complejidad.
2. Discutir si el resultado obtenido en el item anterior contradice la cota inferior $\Omega(n \cdot \lg(n))$
1. Utilizando $\text{Dicc}_{AVL}(\text{Intclave}, \text{Intsignificado})$, en donde las claves serán los números de la lista S y los significados serán la cantidad de veces que aparece el numero en S . Como a lo sumo tendremos $\lfloor \log(n) \rfloor$ valores distintos, la cantidad de elementos del árbol sera esa misma, por lo que su altura sera a lo sumo $\lg(\lg(n))$. Luego, el algoritmo por cada $v \in S$ verificara su existencia en el árbol, si el elemento no existe lo insertara con un significado de 0 y si existe le incrementara 1 a su significado. La complejidad total para los n elementos en este paso sera de $n \cdot \lg(\lg(n))$.

Luego de tener el AVL cargado con cada uno de los elementos, extraeremos el mínimo o máximo (mínimo si ordenamos de menor a mayor, máximo si el orden es el inverso) con un costo de $O(\lg(\lg(n)))$ y siendo k su significado, agregaremos a la lista que devolveremos como resultado k copias del mismo. Repetiremos esta operación hasta que el árbol quede vacío, es decir hasta un máximo de $\lg(\lg(n))$ veces, con el fin de obtener la lista ordenada y con un costo total que podrá ser acotado por $O(n \cdot \lg(\lg(n)))$. Finalmente la complejidad total del algoritmo sera $O(2n \cdot \lg(\lg(n))) \subseteq o(n \cdot \lg(n))$.

2. Si lo hace, de hecho las definiciones mismas ya se contradicen ya que la cota $\Omega(n \cdot \lg(n))$ nos dice por definición que siendo $g(n) = n \cdot \lg(n)$ y $f(n)$ que será la función que representara la complejidad de nuestro algoritmo, para algún n_0 y $c > 0$, valdrá que $c \cdot g(n) \leq f(n)$. Lo que no podrá ser cierto ya que por la definición de o valdrá que $f(n) < d \cdot g(n)$ y juntando ambas tendremos $c \cdot g(n) \leq f(n) < d \cdot g(n)$ lo que no es cierto para todo n , ya que $c \cdot g(n)$ y $d \cdot g(n)$ serán asintóticamente iguales.

Buscando un segundo argumento para obtener la contradicción con Ω , podremos ver que nuestra función sera $f(n) = n \cdot \lg(\lg(n))$ y por la definición de Ω nos quedara que $cn \cdot \lg(n) \leq n \cdot \lg(\lg(n))$, lo que no podrá cumplirse para todo n a partir de algún n_0 y c .

Ejercicio 2

Considerar el TAD diccionario al que se agrega la operación $\text{PrecPrec}(D, k)$, que dado un diccionario D y una clave k devuelve la clave precedente a la precedente a k en D , o un valor especial si no existe.

1. Discutir la complejidad de implementación de $\text{PrecPrec}(D, k)$ en al menos 3 implementaciones eficientes de diccionario.
2. Describir el algoritmo de $\text{PrecPrec}(D, k)$ en la implementación de diccionarios con ABBs.
- 1.
- 2.

Ejercicio 3

Supongamos una secuencia $S = \langle s_1, s_2, \dots, s_n \rangle$ de n enteros distintos (positivos y negativos) en orden creciente. Queremos determinar si existe una posición i tal que $S[i] = i$. Por ejemplo, dada la secuencia $S = \{-4, -1, 2, 4, 7\}$, $i = 4$ es esa posición. Se pide:

1. Diseñar un algoritmo Divide and Conquer eficiente que resuelva el problema
2. Explicar por que el algoritmo propuesto es correcto.

1. El algoritmo tomara como parámetros una cota menor a , una cota mayor b y una secuencia S . Al comienzo de la iteración con S completo, $a = 0$ y $b = |S| = n$. Luego, sea $i = \lfloor b/2 + n \rfloor$, los **casos bases** serán si $S[i] = i$ devolveremos *True* y si $S[i] \neq i \wedge a = b$ devolveremos *False*. De lo contrario estaremos en un **caso recursivo**, en donde si tenemos que $S[i] < i$ entonces definiremos $a = i$ y ejecutaremos recursión, de lo contrario si $S[i] > i$ definiremos $b = i$ y ejecutaremos recursión. La complejidad del algoritmo será exactamente igual que la de búsqueda binaria $O(\lg(n))$, ya que el algoritmo es el mismo solo que en vez de comparar por un valor v compararemos por el índice en la posición que nos encontremos verificando.
2. Sea $j < S[j]$, suponiendo que el arreglo no tiene elementos repetidos (lo que significara que la lista de números sera estrictamente creciente) se cumplirá que $i < S[i]$ para todo $i \in [j, n]$. Por lo tanto de existir algún $i \in [1, n]$ tal que $S[i] = i$, deberá encontrarse en el lado izquierdo del arreglo, es decir que $i \in [1, j)$. El caso simétrico será cuando tengamos $S[j] < j$, en cuyo caso el resultado lo encontraremos a la derecha de j , es decir que $i \in (j, n]$.
– Falta argumentación de porque vale con elementos repetidos –.

Ejercicio 4

Dada la siguiente especificación de un TAD

1. Explicar en palabras las características principales del tipo.
2. Escribir la igualdad observacional.
3. El tipo tal como esta especificado tiene un problema (podríamos decir que esta incorrectamente especificado). Indique cual es y proponga una solución.

Ejercicio 5

Explique cual es la utilidad de las técnicas de eliminación de la recursión. Cuales son las principales ventajas de un algoritmo iterativo respecto a uno recursivo? Es posible que al final del proceso obtengamos un código menos eficiente que el original? Ejemplifique.

4. 1C/2011

Ejercicio 1

Verdadero o Falso, justifique o de un contraejemplo:

1. Si f es $O(g)$ y g es $\Omega(f)$, entonces f es $\Theta(g)$.
2. Si f es $O(n)$, entonces para cualquier entrada f es $\Omega(n)$.
3. Si f es $\Omega(n)$, entonces para cualquier entrada f es $\Theta(n)$.

Ejercicio 2

Sea S una secuencia de inserciones sobre un árbol binario de búsqueda, donde $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ en donde $s_1 \leq s_2 \leq s_3 \dots \leq s_7$, describir las permutaciones de S que formen lo siguiente:

1. Un árbol de altura mínima
2. Un árbol de altura máxima
1. $S = \{s_4, s_2, s_1, s_3, s_6, s_5, s_7\}$ que dará un ABB de altura 3, sera mínimo ya que en un árbol binario completo con n nodos tiene una altura de $\lfloor \lg(n) \rfloor + 1$ y con $n = 7$ esto nos da $\lfloor \lg(n) \rfloor + 1 = 3$.
2. $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ que dará un ABB de altura 7.

Ejercicio 3

Dada la siguiente especificación sobre una cena con participantes, encuentre los errores y corríjalos (axiomatice o describa como arreglar el error)

Tad Persona

generadores:

nueva: edad x dni -> persona

observadores:

. = . persona x persona -> bool

Tad Cena

generadores:

crear: conj(personas) -> cena

llegaInvitado: persona x plato x cena -> cena

observadores:

invitados: cena -> conj(personas)

quePlatoTrajo?: persona p x cena c -> plato (p pertenece invitados(c))

sumaDeEdades: cena -> nat

Tad Regalo es Nat

Ejercicio 4

Se quiere implementar un diccionario sobre una novedosa estructura x , se quiere dividir el trabajo entre 3 personas de la siguiente manera:

1. Escribe la interfaz
2. Escribe algoritmos de inserción y borrado
3. Escribe algoritmos de búsqueda

Que le daría como mínimo de la siguiente lista a cada uno como para que puedan hacer su trabajo, justificar y sea conciso.

- TAD Dicc
- TAD X
- Invariante de representación de dicc sobre x
- invariante de repr de x
- Función de abs de x
- Función de abs de diccionario sobre x
- Interfaz de x

Ejercicio 5

Se quiere implementar conjunto y diccionario cuyas operaciones típicas sean en orden logarítmico (insertar, borrar, buscar o sus equivalentes). De los siguientes diseños, diga cual usaría y porque, si no le convence ninguna proponga una y justifique

1. Dicc y Conjunto sobre AVL, AVL sobre punteros a nodos
2. AVL sobre ABB, ABB sobre AB, AB sobre nodos, Dicc y Conj sobre AVL

5. 1C/2012 (Pombo y Feuerstein)

Ejercicio 1

Había tres códigos distintos que hacían mas o menos lo mismo con leves diferencias, tenias que decir la complejidad de cada uno. Había para todos los gustos: el primero era $O(n^2)$, el segundo era $O(n \lg(n))$ y el tercero era $O(n)$.

Ejercicio 2

Que modificaciones se le puede efectuar al TAD diccionario con el objetivo de poder definir un termino varias veces y agregar una operación de retorno a la definición anterior (que pueda llamarse tantas veces como definiciones anteriores haya)? Sobre que estructura lo implementarías? Dar el pseudocódigo de la operación y la complejidad de todas las operaciones.

Ejercicio 3

Inducción estructural. Dar el esquema de inducción, un ejemplo y explicar bien que es cada cosa separando casos base y casos inductivos. Que particularidad tienen las pruebas de inducción sobre Rosetree?

Ejercicio 4

Invariante de representación. Decir las motivaciones y dar ejemplos.

Ejercicio 5

Eliminación de la recursión. Contar la motivación, decir que tipos de funciones recursivas hay, dar un ejemplo para cada una y decir la complejidad tanto de la versión recursiva como de la imperativa.

6. 2C/2012

Ejercicio 1

Se deben ordenar los parciales de 500 alumnos, en base a su apellido. Para facilitar la practica, los m docentes deciden dividir la tarea entre ellos. Recomiende dos posibles métodos para que utilicen. Tenga en cuenta que:

- Debe ser realizado por personas
- Por mas de una
- Se pretende ademas que sea eficiente (por ejemplo, tratando de que no haya gente sin hacer nada durante mucho tiempo)

Calcule la complejidad de los métodos propuestos.

- Se empezara por repartir $\lfloor 500/m \rfloor$ parciales a cada persona, el ultimo docente al que se le repartirán tendrá suerte ya que probablemente recibirá menos parciales. La siguiente acción sera que cada uno de los docentes ordenara su porción de los parciales en orden lexicográfico y a medida que vayan terminando se moverán de lugar para explicitarlo. Cuando haya al menos dos docentes que hayan terminado con su porción de parciales se organizaran para juntarlos en una sola porción, lo cual haran observando el parcial en el tope de la pila de parciales y formando una nueva pila encolando el menor al final de la misma (en realidad funcionara como una fila) según el orden lexicográfico, esto lo harán hasta que alguno de los dos se quede sin parciales y en dicho momento los parciales restantes irán encolados al final de la pila. Luego, el grupo de dos docentes esperara que se forme otro grupo de dos docentes para juntar sus parciales con el mismo procedimiento, y luego el grupo de cuatro docentes e
- Otra forma sera organizar 26 cajas, una para cada letra con que comience el nombre del apellido, repartir $\lfloor 500/m \rfloor$ parciales a cada persona y que cada docente vaya ubicando los parciales que recibió en la caja correspondiente. Luego

Ejercicio 2

Se tiene un diccionario de idioma castellano representado en un TRIE. Se desea poder manejar palabras acentuadas, de manera tal que cuando se realiza una búsqueda se obtienen los resultados correspondientes a la versión con y sin acento. Por ejemplo se busca "esta" y se obtiene "esta: pronombre..." "esta: verbo estar, primera persona..." etc. Suponga que se cuenta con una función llamada normalizar() que dada una letra acentuada devuelve su equivalente sin acento.

1. Que modificaciones deberían realizarse en la estructura y en los algoritmos para poder contemplar dicha funcionalidad?
2. Este diccionario va a adaptarse para una variante del checo donde son muchos los caracteres que pueden acentuarse. Proponga una variación de la estructura que permita realizar la consulta mencionada en una sola pasada.

Ejercicio 3

Se tiene el siguiente TAD:

```
TAD PUNTO
generadores:
comenzar: -> punto
subir: punto x nat -> punto
derecha: punto x nat -> punto

observadores:
X: punto -> nat
Y: punto -> nat

otras operaciones:
mover: punto x nat n x nat m -> punto

axiomas:
```

```

X(comenzar) = 0
Y(comenzar) = 0
X(subir(p,n)) = X(p)
Y(subir(p,n)) = Y(p)+n
X(derecha(p,n)) = X(p)+n
Y(derecha(p,n)) = Y(p)
mover(p,n,m) = subir(derecha(p,n),m)

```

¿Se puede plantear la demostración por inducción estructural de una propiedad sobre el TAD punto utilizando en los teoremas a demostrar solo `comenzar()`, `mover()`, `X()` e `Y()`? Justifique.

Ejercicio 4

Se diseña un conjunto C de tamaño no acotado sobre una estructura acotada A . Para cada una de las siguientes afirmaciones indique si deberían estar de alguna manera en el invariante de C sobre A , en la función de abstracción de A en C , en ambas o en ninguna. Justifique sus respuestas.

1. Todos los elementos de C están en A
2. Todos los elementos de A están en C
3. En A no hay repeticiones
4. En C no hay repeticiones
5. C no tiene mas elementos que los que entran en A
6. Los elementos de C tienen que tener un orden (por si A es un AVL)
7. Algunas de las características del invariante de A

Ejercicio 5

Especifique el TAD vela, que debe modelar la simulación discreta de una vela. Al comenzar se indica la longitud de la vela en centímetros, que corresponde a la parte cubierta de cera. El cabo mide al comienzo 10 mm. Una vez que se enciende, el cabo disminuye 1 mm por cada pulsación de un reloj. Cuando disminuye el trecho final del cabo (el ultimo milímetro), se derrite instantáneamente 1 cm mas de cera, descubriendo el cabo nuevamente.

TAD Vela

```
igualdad observacional: (\forall v, v': vela) (v =_obs sii
    (
        longitudInicial(v) = longitudInicial(v')
        y
        tiempoTranscurrido(v) = tiempoTranscurrido(v')
    )
)
```

generos: Vela

exporta: Vela, generadores, observadores, velaAcabada?, longCabo?, longCera?, tiempoRestante?

usa: Bool, Nat

generadores:

encender: Nat n -> Vela {n > 0}

pulsacion: Vela -> Vela

observadores:

longitudInicial: Vela -> nat

tiempoTranscurrido: Vela -> nat

otras operaciones:

velaAcabada?: Vela -> bool

longCabo?: Vela -> nat

longCera?: Vela -> nat

tiempoRestante?: Vela -> nat

axiomas:

longitudInicial(encender(n)) = n

longitudInicial(pulsacion(v)) = longitudInicial(v)

tiempoTranscurrido(encender(n)) = 0

tiempoTranscurrido(pulsacion(v)) = 1 + tiempoTranscurrido(v)

velaAcabada?(v) = tiempoTranscurrido(v) >= (longitudInicial(v)+1)*10

longCabo?(v) = if velaAcabada?(v) then 0 else tiempoTranscurrido(v) mod 10 fi

longCera?(v) = if velaAcabada?(v) then 0

else longitudInicial(v)-floor(tiempoTranscurrido(v)/10) fi

tiempoRestante?(v) = longCabo?(v) + longCera?(v)*10

7. 07/02/2013

Ejercicio 1

1. El invariante de representación suele escribirse de manera formal y eso permite utilizarlo para una serie de cosas. Si se escribiese en castellano, ¿cuales de esas cosas se podrían seguir haciendo y cuales no? Justifique.
2. Si el invariante de un tipo resultase programable, ¿lo haría? ¿para que lo usaría? Justifique.
1. En primer lugar el invariante escrito en castellano podría presentar ambigüedades por la naturaleza del lenguaje, por lo que podríamos tener diferentes interpretaciones de las condiciones que se cumplen sobre la estructura de datos a la hora de usarla y usarla de forma errónea o asumiendo cosas que no son ciertas. Suponiendo en un plano utópico que no presenta ambigüedades y que siempre es interpretado de la misma forma, no podríamos realizar demostraciones formales acerca de la correctitud de las funciones, es decir, verificar formalmente si las funciones mantienen el invariante.
2. Bajo mi punto de vista, solamente tiene sentido implementarlo si en la implementación tenemos algun constructor que usa directamente una instancia de la estructura de representación para construir la instancia del modulo. De esta forma con un invariante de representación implementado podríamos verificar que la instancia de la estructura que nos pasan como parámetro es una instancia valida. Otro caso en donde puede ser útil la implementación del invariante es para verificar que las operaciones funcionen correctamente exponiéndolas a distintos escenarios, es decir para testear nuestro modulo.

Ejercicio 2

El siguiente algoritmo, dado un array A determina si el mismo contiene 3 elementos x, y, z que forman una terna pitagórica (es decir, tales que $x^2 + y^2 = z^2$)

```
FIND-PITNUM(a)
1 for i = 1 to length[a] do
2   for j = 1 to length[a] do
3     for k = 1 to length[a] do
4       if a[i]^2 + a[j]^2 = a[k]^2 then
5         return true
6 return false
```

1. Analice la complejidad del algoritmo anterior.
2. Proponer un algoritmo diferente y analizar su complejidad, que debe ser estrictamente menor que el algoritmo anterior. Para elaborar su solución puede utilizar la siguiente función:

```
EXACT-SUM2(A, n, x)
1 i <- n
2 j <- 1
3 while j < i do
4   if A[i]^2 + A[j]^2 = x then
5     return true
6   else
7     if A[i]^2 + A[j]^2 < x then
8       j <- j + 1
9     else
10      i <- i - 1
11 return false
```

1. La complejidad del algoritmo es $O(n^3)$ en el peor caso en donde n es la cantidad de elementos del arreglo. El peor caso se da cuando no existe una terna en el arreglo que cumpla la condición.
2. Una posible solución de esto sin utilizar la función y no in-place, seria llenar un AVL con todos los elementos del arreglo al cuadrado. Luego con un doble for compararíamos cada una de las combinaciones. La complejidad del algoritmo seria de $O(n^2 \cdot \lg(n))$. Otra posible solución con igual complejidad pero in-place seria ordenar el arreglo llevando esto $O(n \cdot \lg(n))$ y luego probar todos los pares posibles teniendo así n^2 pares utilizando una búsqueda binaria siendo $x = \sqrt{A[i]^2 + A[j]^2}$ el elemento a buscar.

Ejercicio 3

Responda y justifique:

1. Una función recursiva a la cola, ¿siempre puede transformarse en iterativa?
2. Y una función con recursión múltiple?

Ejercicio 4

A continuación se muestran unos breves enunciados junto a fragmentos de su especificación como TADs. Indicar si dichos fragmentos son correctos o presentan errores, y si es así, cuales y por que.

1. En un frasco aislado en un laboratorio se cuenta con una cantidad de hielo, que al ser sometido a calor se transforma en agua, a razón de un dm^2 por caloría. Como el frasco esta aislado, esa es la única transformación posible.

```
TAD frasco
generadores:
    nuevo_frasco: tamano -> frasco
observadores:
    volumen_hielo: frasco -> nat
    volumen_agua: frasco -> nat
operaciones:
    aportar_calorias: frasco x nat -> frasco
    disminuir_hielo: frasco x nat -> frasco
    incrementar_agua: frasco x nat -> frasco
Fin TAD
```

2. Una bolita se desplaza sobre una recta a medida que recibe impulsos. Los impulsos se miden en kilos, y la relación entre ambos es que por cada kilo la bolita avanza dos metros.

```
[...]
observadores:
    posicion: bolita -> nat
operaciones:
    empujar: bolita x nat i x distancia d -> bolita {d = 2i}
[...]
```

3. Se desea modelar una pila que siempre permite aplicar la operación `top()`, pero indica cuando no hay ningún elemento valido. observadores:

```
top: pila -> <elem, bool>
```

4. Idem anterior:

```
top: pila -> indicador
hay_elem?: indicador -> bool
elem: pila x indicador i -> elem {hay_elem?(i)}
```

Ejercicio 5

Supongamos que extendemos el TAD Diccionario agregándole una operación `RestriccionDeRango(x,y)` con $x \leq y$, que elimina del diccionario todos los valores que son mayores que y o menores que x . Por ejemplo, si las claves son reales, luego de una operación `RestriccionDeRango(x,y)` el diccionario contendría solamente las claves en el intervalo cerrado $[x,y]$. Proponga una estructura de datos para implementar eficientemente tanto las operaciones tradicionales de diccionario como la nueva operación, considerando los siguientes casos:

1. Se puede asumir que x e y están en el diccionario.
2. No se puede asumir que x e y están en el diccionario.

1. La primera opción podría ser un AVL aumentado con una lista enlazada entre sus nodos. Para ser posible esta modificación en primer lugar tendremos dos datos satélite en cada nodo, un puntero dirigido a su predecesor y otro a su antecesor (de forma tal de tener una lista doblemente enlazada). Durante la inserción deberíamos obtener el sucesor y el antecesor, el sucesor será el mínimo de los mayores predecesores de nuestro nuevo nodo x y el antecesor será el máximo de los menores predecesores. Para obtenerlos en cada paso durante la inserción compararemos y guardaremos si corresponde en los datos satélites de x los punteros al sucesor y antecesor. Una vez insertado x en el lugar los punteros a antecesor y predecesor estarán correctamente definidos, lo único que restaría hacer es modificar el puntero de predecesor en el antecesor de x , apuntándolo a x , como así también modificar el puntero de antecesor en el predecesor de x . Una vez realizado esto se proseguirá con los balanceos normales del AVL. Para la eliminación se ejecutarán los procedimientos normales que ejecutaríamos en una doble lista enlazada, y luego los balanceos del AVL. Las operaciones adicionales para construir y borrar la lista serán todas $O(1)$, por lo que la inserción y eliminación seguirán teniendo $O(\lg(n))$ en el peor caso, la búsqueda no se vera alterada por lo que seguirá siendo $O(\lg(n))$ también.

Para la operación de RestriccionDeRango buscaremos el mínimo elemento del AVL (yéndonos siempre por la rama izquierda hasta encontrar un NIL de hijo izquierdo) y una vez encontrado borraremos elementos utilizando la doble lista enlazada para movernos hasta que la condición $e < x$ no sea cierta. Una vez borrados todos los elementos menores a x , borraremos todos los elementos mayores a y buscando el maximo y borrando utilizando la doble lista enlazada hasta que la condición $y < e$ no se cumpla, en donde e es el elemento actual en donde nos encontremos iterando. La operación tendrá una complejidad de $O((\#elemMen + \#elemMay) \cdot \lg(n))$, en donde $\#elemMen$ es la cantidad de elementos e que pertenecen al AVL y cumplen $e < x$ y $\#elemMay$ es la cantidad de elementos que cumplen $y < e$. El en el peor caso borraremos todo el árbol, lo que nos dara una complejidad de $O(n \cdot \lg(n))$. Otra opción sin utilizar la doble lista enlazada sera, luego de cada borrado, volver a buscar el mínimo o el máximo desde la raíz lo que tardara $\lg(n)$, en este caso la complejidad asintótica sera la misma pero tendremos una constante mas de $\lg(n)$ proveniente de la búsqueda de cada nuevo mínimo.

Alternativamente a utilizar un AVL podríamos utilizar un Splay Tree, en el cual buscaríamos el elemento x , haríamos Splaying sobre el mismo y luego borraríamos todo el sub-árbol izquierdo ya que por el invariante de los ABB, el sub-árbol izquierdo contendrá claves menores a x (suponiendo que no hay claves repetidas), repetiríamos la misma operación pero ejecutando Splaying sobre y y borrando el sub-árbol derecho. Para borrar cada uno de los sub-árboles utilizaremos $\#elemMen$ operaciones para el sub-árbol izquierdo y $\#elemMay$ para el sub-árbol derecho. En los Splay Tree podremos garantizar que una secuencia de m operaciones consume tiempo $O(m \cdot \lg(n))$. Siendo que borraremos $\#elemMen + \#elemMay$ elementos, tendremos una complejidad de $O((\#elemMen + \#elemMay) \cdot \lg(n))$ amortizada para la operación, aunque podremos hacerlo mas rápidamente si nos olvidamos de realizar Splaying a medida que borramos los elementos de un sub-árbol entero. La complejidades de las demás operaciones del diccionario serán $O(\lg(n))$ amortizado para todas.

Si tenemos valores acotados una opción viable es la del uso de un árbol binario digital. En el mismo, a medida que busquemos el elemento x mediante su representación binaria, borraremos en cada paso todo lo que se encuentre en el sub-árbol izquierdo a medida que avanzamos por sus bits, si es que continuamos por el sub-árbol derecho. Ya que como la longitud de los números estará acotada por b bits, tardaremos tiempo $O(b \cdot \#elemMay)$ en borrar las claves que correspondan. Podremos hacer lo mismo con y pero borrando los sub-árboles derechos si es que continuamos por el sub-árbol izquierdo a medida que avanzamos en su representación binaria. La complejidad total de la operación RestriccionDeRango sera de $O(b \cdot (\#elemMen + \#elemMay))$, las complejidades de eliminación, inserción y búsqueda serán $O(b)$.

2. Podremos utilizar todas las variantes presentadas anteriormente, con leves o ninguna modificación. En la variante del AVL puede realizarse lo mismo sin importar si el elemento esta o no esta en el diccionario, en la variante del árbol binario digital bastara hasta encontrar el elemento o un NIL y en la variante del Splay Tree es importante que x e y existan, pero se podrá solucionar fácilmente insertándolos (si es que no existen) como nodos centinelas antes del borrado y borrándolos luego si es que no se encontraban.